Freie Universität Berlin

Volker Roth

Rechnersicherheit, SoSe 21

Übung 03

TutorIn: Oliver Wiese
Tutorium 02
Materialien: Latex, VSC, Skript

19. Mai 2021

---

1 Markov Generator

(a) *Construct a first-order Markov model for this sample.*

- We implemented a simply python code to construct a first and second order Markov model for this sample. Here is a snippet of the code:

```
import json

def markov(alphabet, database):
    #prepare table
    table = {}
    count = {}
    for char in alphabet:
        count[char] = 0
        for char2 in alphabet:
            table[char+char2] = 0

    #con markox model
    for data in database:
        for i in range(len(data)-1):
            table[data[i]+data[i+1]] +=1

    #count occurrence
    for data in database:
        for i in range(len(data)):
            count[data[i]] +=1

    for char in alphabet:
        for char2 in alphabet:
            if table[char+char2] != 0:
                table[char+char2] /= count[char]
            else:
                table.pop(char+char2, None)

    return table
```

- And here is the result:

```
{
    "s1": 0.6,
    "s2": 0.4,
    "01": 0.16666666666666666,
    "03": 0.5,
    "0t": 0.3333333333333333,
    "10": 0.1111111111111111,
    "12": 0.2222222222222222,
    "13": 0.4444444444444444,
    "1t": 0.2222222222222222,
    "20": 0.2222222222222222,
    "23": 0.5555555555555556,
    "2t": 0.2222222222222222,
    "30": 0.1875,
    "31": 0.125,
    "32": 0.1875,
    "33": 0.25,
    "3t": 0.25
}{
```


(b) *Construct a second-order Markov model for this sample*


```
}{
    "s10": 0.16666666666666666,
    "s12": 0.16666666666666666,
    "s13": 0.6666666666666666,
    "s23": 1.0,
    "01t": 1.0,
    "033": 0.3333333333333333,
    "03t": 0.6666666666666666,
    "103": 1.0,
    "120": 0.5,
    "12t": 0.5,
    "130": 0.25,
    "131": 0.25,
    "133": 0.5,
    "203": 0.5,
    "20t": 0.5,
    "230": 0.2,
    "232": 0.4,
    "233": 0.2,
    "23t": 0.2,
    "301": 0.3333333333333333,
    "303": 0.3333333333333333,
    "30t": 0.3333333333333333,
    "312": 0.5,
    "31t": 0.5,
    "320": 0.3333333333333333,
    "323": 0.3333333333333333,
    "32t": 0.3333333333333333,
    "330": 0.25,
    "331": 0.25,
    "332": 0.25,
    "33t": 0.25
}
```


(c) *Give one 4-digit PIN number that is generated by your first-order Markov model but not by your second-order model and calculate it's probability.*

- 4-digit PIN: 1233 as 123 is not reachable in the second Markov model.
- Probability: $\frac{6}{10} \cdot \frac{2}{9} \cdot \frac{4}{8} \cdot \frac{4}{16} = \frac{192}{11520} = \frac{1}{60}$

2 Project

(a) *A client can create a new user and chose a password. You do not have to implement password reset or changing the password.*

(b) *A user can login with a given username and password.*

(c) *Only authenticated clients can send or receive messages.*

- Authentication code for server:

```
def authenticate(client_socket, lock):

    #recv credentials
    login_or_create, username, password = client_socket.recv(DATASIZE).decode('utf-8').split('\
    t')

    #load credentials data
    credentials_data = {}
    try:
        with open(credentials_file, 'rb') as file:
            credentials_data = pickle.load(file)
    except:
        pass

    if login_or_create == 'login':
        #test if login credentials exist
        try:
            password_pepper = password.encode('utf-8') + pepper
            #Success
            if bcrypt.checkpw(password_pepper, credentials_data[username]):
                client_socket.send(b'1')
                return 1
            else:
                client_socket.send(b'Wrong credentials')
                sys.exit()
        except:
            client_socket.send(b'Account does not exist')
            sys.exit()

    if login_or_create == 'create':
        #lock because if 2 users create an account with the same username at the same time
        #one account will be overwritten
        with lock:
            try:
                #check if username is already in use
                credentials_data[username]
                client_socket.send(b'Username already in use, please try again')
                sys.exit()

            except:
                #hash password with pepper and salt
                password_pepper = password.encode('utf-8') + pepper
                hashed = bcrypt.hashpw(password_pepper, bcrypt.gensalt())

                #upload password hash and salt to credentials data
                credentials_data[username] = hashed

                #save updatet credentials
                with open(credentials_file, 'wb') as file:
                    pickle.dump(credentials_data, file)

                #send success msg to client
                client_socket.send(b'1')
    else:
        raise Exception("An error in authentication has occurred")
```

- Authentication code for client:

```python
def authenticate(server_socket):
    login_or_create = input("Enter 0 for login or 1 to create a new account: ")

    #login
    if(login_or_create == '0'):
        username = input("Please enter your username: ")
        password = input("Please enter your password: ")

        #tell server that clients wants to login
        response = f'login\t{username}\t{password}'
        #ask server if account matches credentials
        server_socket.sendall(response.encode('utf-8'))
        response = server_socket.recv(DATASIZE)

        #if successful
        if response == b'1':
            print('Success, you are logged in!')
            return 1

        else:
            print(response.decode('utf-8'))
            sys.exit()

    #create
    if(login_or_create == '1'):
        username = input("Please enter a username: ")
        password = input("Please enter a password: ")

        #tell server that clients wants to create an account
        response = f'create\t{username}\t{password}'
        #ask server if account credentials are free
        server_socket.sendall(response.encode('utf-8'))
        response = server_socket.recv(DATASIZE)

        #if successful
        if response == b'1':
            print('Success, you are logged in!')
            return 1

        else:
            print(response.decode('utf-8'))
            sys.exit()

    else:
        print('Wrong Input')
        sys.exit()
```

```
ocker-compose up -d --force-recreate
Recreating projekt_server_1 ... done
Recreating projekt_client_1 ... done
Recreating projekt_client_2 ... done
Recreating projekt_client_3 ... done
```

```
ocker attach projekt_client_1
1
Please enter a username: this_creates_a_new_account
Please enter a password: password1
Success, you are logged in!
```

```
ocker attach projekt_client_2
0
Please enter your username: now_I_try_to_log
Please enter your password: but_this_account_doesent_exists
Account does not exist
```

(d) *The password-information are stored in a file.*

- We stored them using the pickle module. The modules stores data in bytes, so the file containing the password-information is 'sadly' not readable by humans.

But before implementing your password-based authentication you should document the following problems:

(a) *Think about possible pitfalls when implementing password-based authentication. List all of your pitfalls.*

(b) *Describe briefly how to avoid your pitfalls.*

- Pitfalls:
  1. The account credentials have to be stored. When some hacker gets access to the file containing the credentials. He would have access to all the accounts (passwords, usernames .etc)
     - How we avoided it: We didn't store the passwords plain in a txt file, but we hashed them.

2. Rainbow tables can be used.
   - How we avoided it: Before hashing the passwords we made use of a salt, and saved the salt afterwards together with the password in the credentials file.
3. The hacker could just brute force it (like we did in exercise 2) or use a dictionary attack.
   - How we avoided it: We used a random hash for every password, increasing the time to brute force it by a lot.
4. Even if it takes long, it can still be brute forced (or a dictionary attack can be used).
   - How we avoided it: Instead of just using salt, we also used pepper, which is not stored with the file, which makes the chance to crack the password negligible
5. Instead of trying to steal the password file, the hacker could just brute force / dictionary attack our authentication (login) service.
   - How we are going to avoid it (not yet implemented): We will setup a strong password policy.
   - We will limit the amount of times one user (IP) can try to login.

(c) *If you searched for password-based authentication, describe your search and results. What was helpful and what was dangerous?*

- Helpful:
  - Many sources advised to use bcrypt instead of SHA256-crypt, as it is stronger against brute force.
- Dangerous:
  - Some sources advised to use an unsafe kind to store passwords. Like storing them plain or just with hash without a salt and pepper.